Parallel I/O

Giorgio Amati
g.amati@cineca.it

Roma, 22 July 2016

Agenda

- ✓I/O: main issues
- Parallel I/O: main issues
- Some examples
- Comments

Some questions

- ✓ Which is the typical I/O size you work with?
  - GB?
  - TB?
- ✓ Is your code parallelized?
- ✓ How many cores are you using?
- ✓ Are you working in a small group or you need to exchange data with other researchers?
- ✓ Ever faced I/O problems?
- ✓ Blocksize ? RAID?

I/O: some facts

I/O is a crucial issue in modern HPC applications:

- ✓ deal with very large datasets while running massively parallel
- ✓ applications on supercomputers
- ✓ amount of data saved is increased
- ✓ latency to access to disks is not negligible
- ✓ data portability (e.g. endianness)

HW solution: parallel filesystem (gpfs, lustre, ….)
SW solution: high level libraries (MPI I/O, HDF5,)

Keep in mind: I/O is very very very slow!!!!

“Golden” rules about I/O

- Reduce I/O as much as possible: only relevant data must be stored on disks
- Save data in binary/unformatted form:
  - ✓ asks for less space comparing with ASCI/formatted ones
  - ✓ It is faster (less OS interaction)
- Save only what is necessary to save for restart or check-pointing, everything else, unless for debugging or quality check, should be computed on the fly.
- Dump all the quantities you need once, instead of using multiple I/O calls: if necessary use a buffer array to store all the quantities and the save the buffer using only a few I/O calls.
- Why?

What is I/O?

- ✓ Raw data (in RAM)
- ✓ **`fwritef, fscanf, fopen, fclose, WRITE, READ, OPEN, CLOSE`**
- ✓ Call to an external library: OS, MPI I/O, HDF5, NetCDF…
- ✓ Scalar/parallel/network Filesystems
  1. I/O nodes and Filesystem cache
  2. I/O network (IB, SCSI, Fibre, ecc..)
  3. I/O RAID controllers and Appliance (Lustre, GPFS)
  4. Disk cache
  5. FLASH/Disk (one or more Tier)
- ✓ …eventually write on tape

Latencies

- I/O operations involves
  - ✓ OS & libraries
  - ✓ IO devices (e.g. RAID controllers)
  - ✓ Disks
- I/O latencies of disks are of the order of microseconds
- RAM latencies of the order of 100-1000 nanoseconds
- FP unit latencies are of the order of 1-10 nanoseconds
- → I/O very very very slow compared to RAM of FP latencies

I/O Some figures

- ✓ Real word CFD code
- ✓ Time to dump
- ✓ Serial performance
- ✓ Marconi gpfs Filesystem

Size	Time (sec)	MB/s
20 MB	0.0715″	280
65 MB	0.15″	433
153 MB	0.25″	612
514 MB	0.58″	886
1.2 GB	1.5″	820
4.1 GB	4.2″	999
9.6 GB	9.6″	1024
33 GB	35″	965

Architectural trends/1

		2020 estimates
Number of cores	↑	10^9
Memory x core	↓	100Mbyte or less
Memory BW/core	↓	500GByte/sec
Memory hierachy	↑	Reg, L1, L2, L3, …

Architectural trends/2

	2020 estimates
Wire BW/core	1GByte/sec
Network links/node	100
Disk perf	100Mbyte/sec
Number of disks	100K

I/O: ASCII vs. binary/1

- ASCII is more demanding respect binary in term of disk occupation
- Numbers are stored in bit (single precision floating point number → 32 bit)
- 1 single precision on disk (binary) → 32 bit
- 1 single precision on disk (ASCII) → 80 bit
  - 10 or more `char` (`1.23456e78`)
  - Each char asks for 8 bit

✓ Not including spaces, signs, return, …

✓ Moreover there are rounding errors, …

I/O: ASCII vs. binary/2

- Some figures from a real world application (**openFOAM**)
- Test case: 3D Lid Cavity, 200^3, 10 dump

- Formatted output (ascii)
  - ✓ Total occupation: 11 GB
- Unformatted output (binary)
  - ✓ Total occupation: 6.1 GB
- A factor 2 in disk occupation!!!!

I/O: blocksize

- The blocksize is the basic (atomic) storage size
- One file of 100 bit will occupy 1 blocksize, that could be > 4MB

```
ls -lh TEST_1K/test_1
-rw-r--r-- 1 gamati01  10K 28 gen 11.22 TEST_1K/test_1
…
du -sh TEST_1K/test_1
512KTEST_0K/test_1
…
du -sh TEST_1K/
501M TEST_10K/
…
```

- Always use **tar** commando to save space

```
-rw-r--r-- 1 gamati01 11M  5 mag 13.36 test.tar
```

I/O: endianess

- IEEE standard set rules for floating point operations
- But set no rule for data storage
- Single precision FP: 4 bytes (**B0**,B1,B2,B3)
  - ✓ Big endian (IBM): **B0** B1 B2 B3
  - ✓ Little endian (INTEL): B3 B2 B1 **B0**
- Solutions:
  - ✓ Hand made conversion
  - ✓ Compiler flags (intel, pgi)
  - ✓ I/O libraries (HDF5)

Agenda

- ✓I/O: main issues
- ✓Parallel I/O: main issues
- Some examples
- Comments

What is parallel I/O?

- Serial I/O
  - ✓ Master task writes all the data
- Parallel I/O
  - ✓ Distributed IO on local files: tasks write its own data in a different file
  - ✓ high level libraries: MPI/IO, HDF5, NetCDF, CGNS

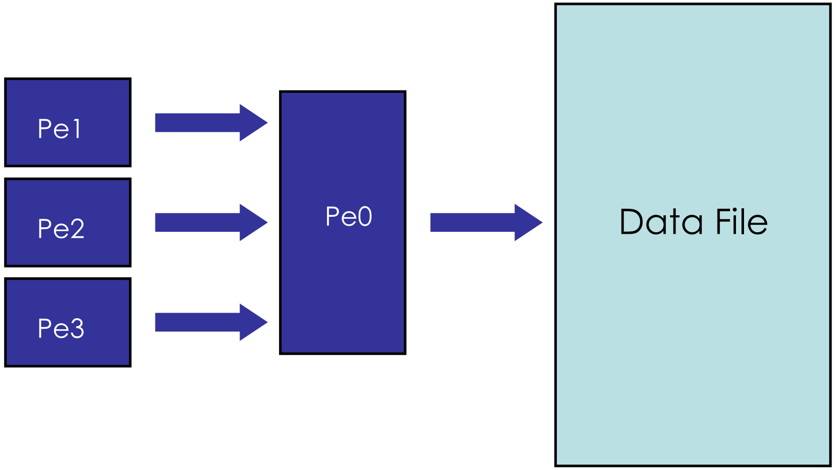
No performance gain if thers's no parallel filesystem!!!!

Why parallel I/O?

- New Architectures: many-many core (up to 10^9)
- As the number of task/threads increases I/O overhead start to affect performance
- I/O (serial) will be a serious bottleneck
- Parallel I/O is mandatory else no gain in using many-many core
- Other issues:
  - ✓ domain decomposition
  - ✓ data management

Managing I/O in Parallel Applications

Master-Slave approach: only 1 processor performs I/O



No scalability

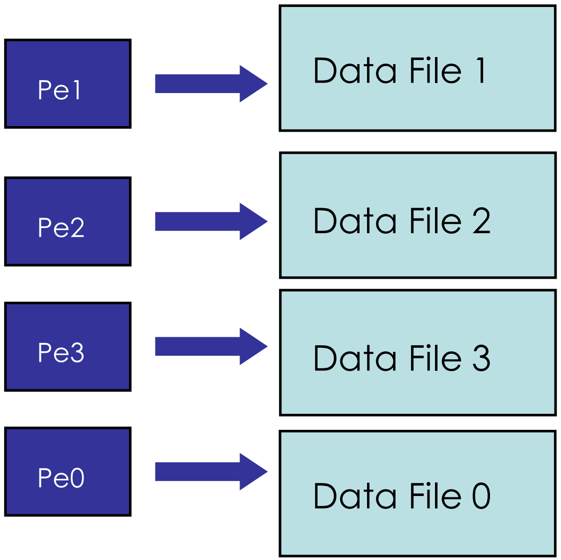
Extra communications

Usability

no parallel FS needed

Managing I/O in Parallel Applications

Distributed IO on local files: all the processors read/writes their own files



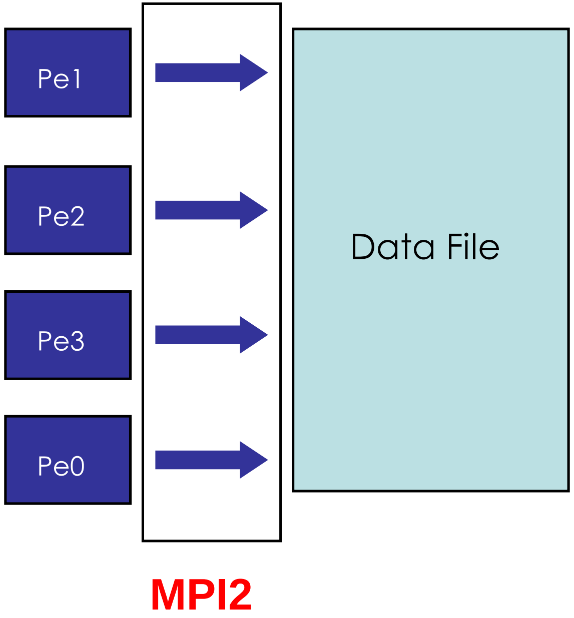
Scalable

No extra communication

Usability

Managing IO in Parallel Applications

I/O Library (MPI I/O or other) : MPI functions perform the IO. Asynchronous IO is also supported.

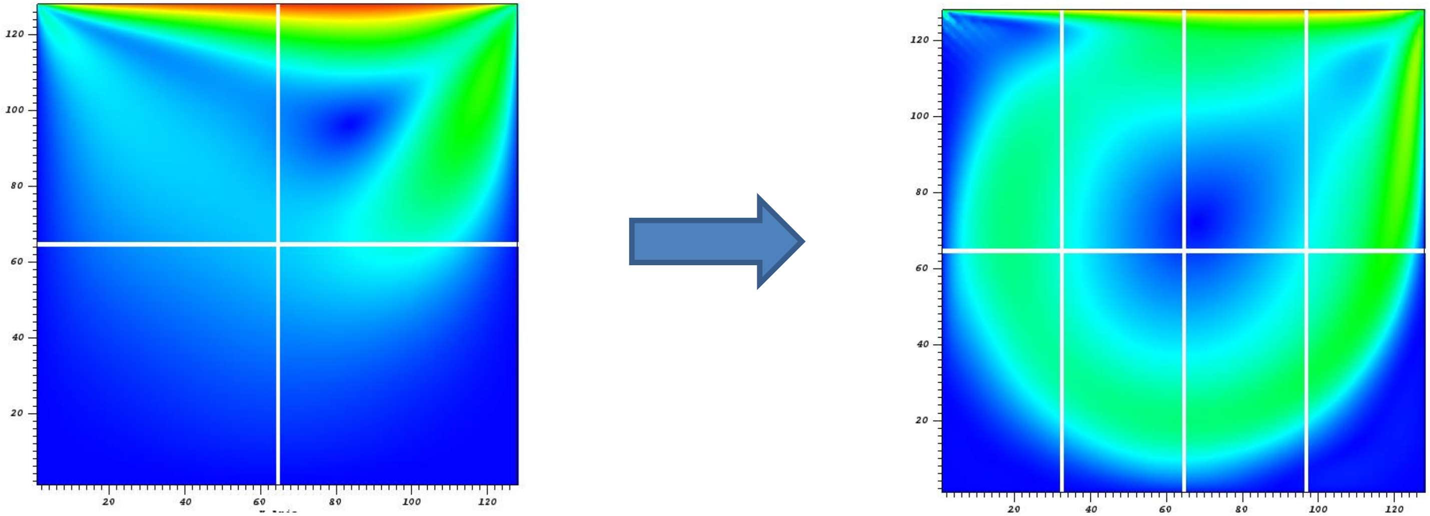


Scalable

Avoid communication

Usability

I/O: Domain Decomposition



- I want to restart a simulation using a different number of tasks: three possible solutions
  - ✓ pre/post processing (merging & new decomposition)
  - ✓ serial dump/restore (memory limitation)
  - ✓ Parallel I/O (single restart file)

Some figures/1

- Simple CFD program, just to give you an idea of performance loss due to I/O.
- 2D Driven Cavity simulation: size 2048*2048, double precision (about 280 MB), 1000 timestep
- Serial I/O = 1.5''
  - ✓ 1% of total serial time
  - ✓ 16% of total time using 32 Tasks (2 nodes) → 1 dump ≈ 160 timestep
- Parallel I/O = 0.3'' (using MPI I/O)
  - ✓ 3% of total time using 32 Tasks (2 Nodes) → 1 dump ≈ 30 timestep
- And using 256 or more tasks?

Some figures/2

- Performance to dump huge files using Galileo: same code with different I/O strategies….
- RAW (512 files, 2.5GB per file)
  - ✓ Write: 3.5 GB/s
  - ✓ Read: 5.5 GB/s
- HDF5 (1 file, 1.2TB)
  - ✓ Write: 2.7 GB/s
  - ✓ Read: 3.1 GB/s
- MPI-IO (19 files, 64GB per file)
  - ✓ Write: 3.1 GB/s
  - ✓ Read: 3.4 GB/s

Some figures/3

- ✓Parallel performance/HDF5
- ✓Marconi Filesystem

Size	1 task	2 task	4 task	8 task	16 task
33 GB	.99 GB/s	1.8 GB/s	3.6 GB/s	4.5 GB/s	3.8 GBs

Size	4 task	8 task	16 task	32 task	64 task
77 GB	2.1 GB/s	4.8 GB/s	7 GB/s	7.7 GB/s	5.4 GBs

Agenda

- ✓I/O: main issues
- ✓Parallel I/O: main issues
- ✓Some examples
  - ✓An example with I/O
  - ✓Few info about HDF5
- Comments

MPI-2.x: features for Parallel I/O

- MPI-IO: introduced in MPI-2.x standard (1997)
  - ✓ allow non-contiguous access in both memory and file
  - ✓ reading/writing a file is like send/receive a message from a MPI buffer
  - ✓ optimized access for non-contiguous data
  - ✓ collective/non-collective access operations with communicators
  - ✓ blocking/non-blocking calls
  - ✓ data portability (implementation/system independent)
  - ✓ good performance in many implementations
- Why do we start to use it ???
  - syntax and semantic are very (???) simple to use

Starting with MPI-I/O

- MPI-IO provides basic IO operations:
  - ✓ open, seek, read, write, close (etc.)
- open/close are collective operations on the same file
  - ✓ many modalities to access the file
- read/write are similar to send/recv of data to/from a buffer
  - ✓ each MPI process has its own local pointer to the file (individual file pointer) for seek, read, write operations
  - ✓ offset variable is a particular kind of variable and it is given in elementary unit (etype) of access to file (default in byte)
  - ✓ it is possible to know the exit status of each subroutine/function

MPI I/O in a nutshell

- Create the correct datatype
  - ✓ **`MPI_Type_create_subarray`**
  - ✓ **`MPI_Type_commit`**
- Define file offset/size
  - ✓ **`MPI_File_seek`**
  - ✓ **`MPI_File_get_size`**
- define fileview
  - ✓ **`MPI_File_set_view`**
- Write or Read file
  - ✓ **`MPI_File_write/MPI_File_read`**
- File sync (flush any caches/buffer)
  - ✓ **`MPI_File_sync`**

MPI I/O: file positioning

There are different way to file positioning (file access)

- Explicit offset: each task computes explicitly the offset (i.e. the physical starting point of the file where to write/read)
- Individual file point: each task has its own file pointer on the file where to start write/read
- Shared file point: each task share the same file pointer once one task has finisher his work all other tasks know where to write

MPI I/O in a nutshell 2

MPI_FILE_OPEN

- ✓ MPI_MODE_RDONLY: read only
- ✓ MPI_MODE_RDWR: reading and writing
- ✓ MPI_MODE_WRONLY: write only
- ✓ MPI_MODE_CREATE: create the file if it does not exist
- ✓ MPI_MODE_EXCL: error if creating file that already exists
- ✓ MPI_MODE_DELETE_ON_CLOSE: delete file on close
- ✓ MPI_MODE_UNIQUE_OPEN: file will not be concurrently opened elsewhere
- ✓ MPI_MODE_SEQUENTIAL: file will only be accessed sequentially
- ✓ MPI_MODE_APPEND:

MPI_File_close

Data Access

Positioning	Synchronisation	Coordination	
		Noncollective	*Collective*
Explicit offsets	*Blocking*	MPI_FILE_READ_AT MPI_FILE_WRITE_AT	MPI_FILE_READ_AT_ALL MPI_FILE_WRITE_AT_ALL
	Non-blocking & split collective	MPI_FILE_IREAD_AT MPI_FILE_IWRITE_AT	MPI_FILE_READ_AT_ALL_BEGIN MPI_FILE_READ_AT_ALL_END MPI_FILE_WRITE_AT_ALL_BEGIN MPI_FILE_WRITE_AT_ALL_END
Individual file pointers	*Blocking*	MPI_FILE_READ MPI_FILE_WRITE	MPI_FILE_READ_ALL MPI_FILE_WRITE_ALL
	Non-blocking & split collective	MPI_FILE_IREAD MPI_FILE_IWRITE	MPI_FILE_READ_ALL_BEGIN MPI_FILE_READ_ALL_END MPI_FILE_WRITE_ALL_BEGIN MPI_FILE_WRITE_ALL_END
Shared file pointer	*Blocking*	MPI_FILE_READ_SHARED MPI_FILE_WRITE_SHARED	MPI_FILE_READ_ORDERED MPI_FILE_WRITE_ORDERED
	Non-blocking & split collective	MPI_FILE_IREAD_SHARED MPI_FILE_IWRITE_SHARED	MPI_FILE_READ_ORDERED_BEGIN MPI_FILE_READ_ORDERED_END MPI_FILE_WRITE_ORDERED_BEGIN MPI_FILE_WRITE_ORDERED_END

Example: Individual file pointers/1

```
PROGRAM main
      use mpi
      implicit none
      integer, parameter :: filesize=8
!
      integer buf(filesize)
      integer rank,ierr,fh,nprocs,nints,intsize,count,i
      integer status(MPI_STATUS_SIZE)
      integer(kind=MPI_OFFSET_KIND) offset
!
! mpi stuff
      call MPI_INIT(ierr)
      call MPI_COMM_RANK(MPI_COMM_WORLD,rank,ierr)
      call MPI_COMM_SIZE(MPI_COMM_WORLD,nprocs,ierr)
      call MPI_TYPE_SIZE(MPI_INTEGER,intsize,ierr)
!
! set #of elements for task
      count=filesize/nprocs
```

Example: Individual file pointers/2

```
! set file offset for task
      offset=rank*count*intsize
!
      do i=1, count
!         buf(i) = rank*count + i
         buf(i) = rank
      enddo
!
      write(6,*) "Task ", rank, " write ", buf(1), " from ", offset
!
      call MPI_FILE_OPEN(MPI_COMM_WORLD,'out.bin',MPI_MODE_WRONLY+MPI_MODE_CREATE,  &
                         MPI_INFO_NULL,fh,ierr)
      call MPI_FILE_SEEK(fh,offset,MPI_SEEK_SET,ierr)
      call MPI_FILE_WRITE(fh,buf,count,MPI_INTEGER,status,ierr)
      call MPI_FILE_CLOSE(fh,ierr)
      call MPI_FINALIZE(ierr)
END PROGRAM main
```

Example: explicit offset/3

```
      call  MPI_FILE_OPEN(MPI_COMM_WORLD,'out.bin',MPI_MODE_RDONLY, &
                          MPI_INFO_NULL,fh,ierr)
      if(ierr == 0) then
         write(6,*) "file exists....."
      else
         write(6,*) "Huston we have a problem!"
         call MPI_FINALIZE(ierr)
      endif
!
      call MPI_FILE_READ_AT(fh,offset,buf,count,MPI_INTEGER,status,ierr)
      call MPI_FILE_CLOSE(fh,ierr)
      call MPI_FINALIZE(ierr)
END PROGRAM main
```

MPI I/O: some figures

```
gamati01@node001.pico:[SE2016]$ mpirun -n 1 ./MPIwrite.x
 Task               0   write           1   from                 0
 Total IO time    2.035156

gamati01@node001.pico:[SE2016]$ mpirun -n 2 ./MPIwrite.x
 Task               1   write   134217729   from         536870912
 Task               0   write           1   from                 0
 Total IO time    1.203125

gamati01@node001.pico:[SE2016]$ mpirun -n 4 ./MPIwrite.x
 Task               2   write   134217729   from         536870912
 Task               3   write   201326593   from         805306368
 Task               0   write           1   from                 0
 Task               1   write    67108865   from         268435456
 Total IO time   0.7070312
```

MPI I/O: advanced issues

Basic MPI-IO features are not useful when

- Data distribution is non contiguous in memory and/or in the file
  - ✓ ghost cells
  - ✓ block/cyclic array distributions
- Multiple read/write operations for segmented data generate poor performances

MPI-IO allow to access to data in different way:

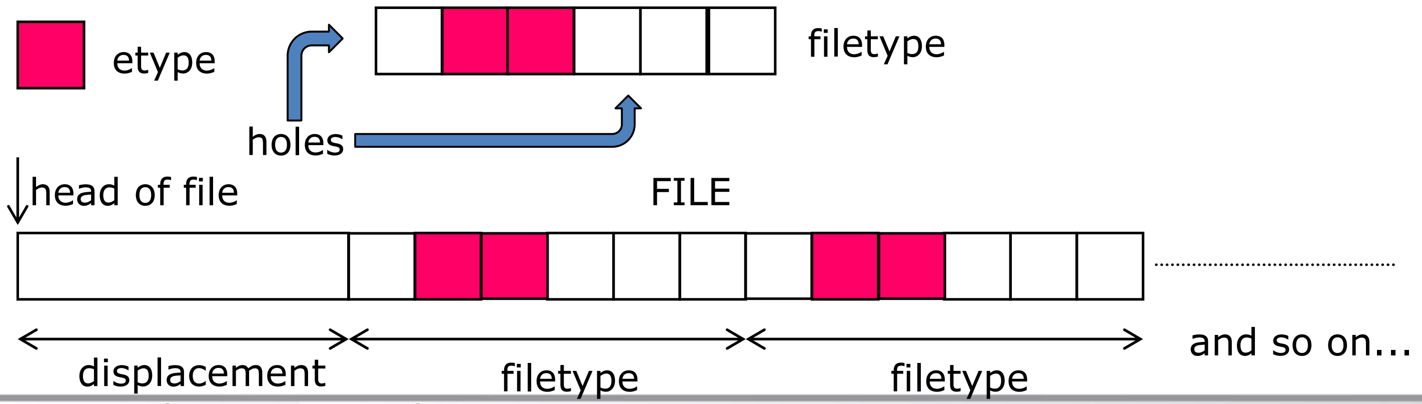
- non contiguous access on file: providing the access pattern to file (fileview)
- non contiguous access in memory: setting new MPI derived datatype

MPI-I/O: File View

A file view defines which portion of a file is "visible" to a process: needs three components

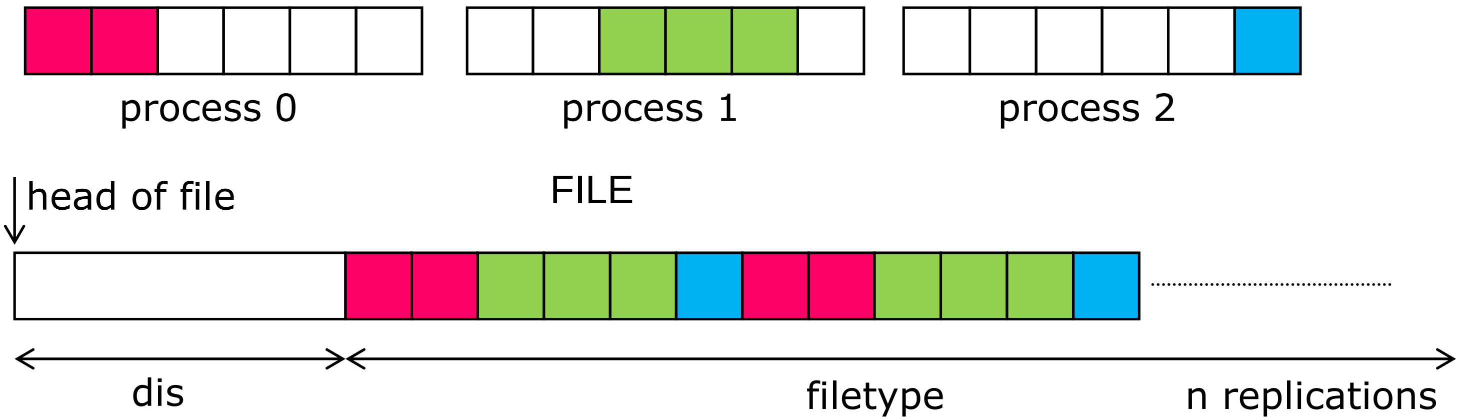
- ✓ **displacement** : number of **bytes** to skip from the beginning of file
- ✓ **etype** : type of data accessed, defines unit for offsets
- ✓ **filetype** : base portion of file visible to a process

The pattern described by a file-type is repeated, beginning at the displacement, to define the view, as it happens when creating MPI_CONTIGUOUS or when sending more than one MPI datatype element: HOLES are important!



MPI I/O: complex pattern

- ✓MPI fileview allow complex replicated pattern access (e.g. struct)



MPI I/O: 3D decomposition/1

```
gsize(1)=lx                   !global size
gsize(2)=ly
gsize(3)=lz
lsize(1)=l                    ! Local size (for each task)
lsize(2)=m
lsize(3)=n
offset(1) = mpicoords(1)*l    ! offset
offset(2) = mpicoords(2)*m
offset(3) = mpicoords(3)*n
buffersize = l*m*n
```

MPI I/O: 3D decomposition/2

```
call MPI_TYPE_CREATE_SUBARRAY(mpid,gsize,lsize,offset,MPI_ORDER_FORTRAN, &
                              MYMPIREAL,dump3d,ierr)
Call MPI_TYPE_COMMIT(dump3d,ierr)

call MPI_FILE_OPEN(LBMCOMM,filename01,MPI_MODE_WRONLY+MPI_MODE_CREATE, &
                   MPI_INFO_NULL,myfile,ierr)
call MPI_FILE_SET_VIEW(myfile,file_offset,MYMPIREAL,dump3d,'native', &
                       MPI_INFO_NULL,ierr)
call MPI_FILE_WRITE_ALL(myfile,buffer,buffersize,MUMPIREAL, &
                        MPI_STATUS_IGNORE, ierr)
```

HFD5: some history…

- **Hierarchical Data Format**
  - ✓ is a set of file formats and libraries designed to store and organize large amounts of numerical data
  - ✓ It is a hierarchical, filesystem-like data format. Resources in an HDF5 file are accessed using the syntax */path/to/resource*. Metadata are stored in the form of user-defined, named attributes attached to groups and datasets
- Originally developed at the NCSA, it is supported by the non-profit HDF Group (www.hdfgroup.org), whose mission is to ensure continued development of HDF5 technologies
- Last HDF5 releases:
  - ✓ 1.10.0 (first release of the new minor revision 1.10)
  - ✓ 1.8.16 (last release of the minor revision 1.8)

HDF5 file

- An HDF5 file is a "*container*" for storing a variety of (scientific) data
- Is composed of two primary types of objects:
  - ✓ Groups: a grouping structure containing zero or more HDF5 objects, together with supporting metadata
  - ✓ Datasets: a multidimensional array of data elements, together with supporting metadata
- Any HDF5 group or dataset may have an associated attribute list
  - ✓ Attribute: a user-defined HDF5 structure that provides extra information about an HDF5 object.

A look inside an hdf5 file

- `h5dump -H u_00001000.h5`

```
HDF5 "u_00001000.h5" {
GROUP "/" {
   GROUP "field" {
      DATASET "rho" {
         DATATYPE  H5T_IEEE_F32LE
         DATASPACE  SIMPLE { ( 64, 1, 64 ) / ( 64, 1, 64 ) }
      }
      DATASET "u" {
         DATATYPE  H5T_IEEE_F32LE
         DATASPACE  SIMPLE { ( 64, 1, 64 ) / ( 64, 1, 64 ) }…
```

Agenda

- ✓I/O: main issues
- ✓Parallel I/O: main issues
- ✓Some examples
- ✓Comments

I/O: managing data

- TB of different data sets
- Hundreds of different test cases
- Metadata
- Share data among different researchers
  - ✓ different tools (e.g. visualization tools)
  - ✓ different analysis/post processing

- You need a common "language"
  - ✓ Use I/O libraries
  - ✓ Invent your own data format

Some strategies

- I/O is the bottleneck → avoid when possible
- I/O subsystem work with locks → simplify application
- I/O has its own parallelism → use MPI-I/O
- I/O is slow → compress (to reduce) output data
- Raw data are not portable → use library
- I/O C/Fortran APIs are synchronous → use dedicated I/O tasks

- Application DATA are too large → analyze it "on the fly", (e.g. re-compute vs. write)

At the end: moving data

- Now I have hundreds of TB. What I can do?
  - ✓ Storage using Tier-0 Machine is limited in time (e.g. PRACE Project data can be stored for 3 Month)
  - ✓ Data analysis can be time consuming (eyen years)
  - ✓ I don’t want to delete data
  - ✓ I have enough storage somewhere else?

  ✓ **How can I move data?**

Moving data: theory

- BW requirements to move Y Bytes in Time X

Bits per Second Requirements

	1H	8H	24H	7Days	30Days
10PB	25,020.0 Gbps	3,127.5 Gbps	1,042.5 Gbps	148.9 Gbps	34.7 Gbps
1PB	2,502.0 Gbps	312.7 Gbps	104.2 Gbps	14.9 Gbps	3.5 Gbps
100TB	244.3 Gbps	30.5 Gbps	10.2 Gbps	1.5 Gbps	339.4 Mbps
10TB	24.4 Gbps	3.1 Gbps	1.0 Gbps	145.4 Mbps	33.9 Mbps
1TB	2.4 Gbps	305.4 Mbps	101.8 Mbps	14.5 Mbps	3.4 Mbps
100GB	238.6 Mbps	29.8 Mbps	9.9 Mbps	1.4 Mbps	331.4 Kbps
10GB	23.9 Mbps	3.0 Mbps	994.2 Kbps	142.0 Kbps	33.1 Kbps
1GB	2.4 Mbps	298.3 Kbps	99.4 Kbps	14.2 Kbps	3.3 Kbps
100MB	233.0 Kbps	29.1 Kbps	9.7 Kbps	1.4 Kbps	0.3 Kbps

Moving data: practice/1

- Moving outside CINECA
  - ✓ **`scp`** → 10 MB/s
  - ✓ **`rsync`** → 10 MB/s
- I must move 50TB of data:
  - ✓ Using **`scp`** or **`rsync`** → 60 days

- No way!!!!!
- Bandwidth depends on network you are using. Could be better, but in general is even worse!!!

Moving Data: practice/2

- Moving outside CINECA
  - **`gridftp`** → 100 MB/s
  - **`globusonline`** → 100 MB/s
- I must move 50TB of data:
  - Using **`gridftp/globusonline`** → 6 days
- Could be a solution…

- Note
  - We get these figures between CINECA and a remote cluster using a 1Gb Network

Moving Data: some hints

- **Size matters**: moving many little files cost more then moving few big files, even if the total storage is the same!
- Moving file from Fermi to a remote cluster via Globusonline

Size	Num. Of files	Mb/s
10 GB	10	227
100 MB	1000	216
1 MB	100000	61

✓ You can loose a factor 4, now you need 25 days instead of 6 to move 50TB!!!!!

Moving Data: some hints

- ✓ Plan your data-production carefully
- ✓ Plan your data-production carefully (again!)
- ✓ Plan your data-production carefully (again!!!!!)
- ✓ Clean your dataset from all unnecessary stuff
- ✓ Compress all your ASCII files
- ✓ Use **`tar`** to pack as much data as possible
- ✓ Organize your directory structure carefully
- ✓ Syncronize with **`rsync`** in a systematic way
- ✓ One example:
  - We had a user who wants to move 20TB distributed over more then 2’000’000 files…
  - **`rsync`** asks many hours (about 6) only to build the file list, without any synchronization at all

Best Practices

- When designing your code, think I/O carefully!
  - ✓ maximize the parallelism
  - ✓ if possible, use a single file (of few) as restart file and simulation output
  - ✓ avoid the usage of formatted output (do you actually need it?)
- Minimize the latency of file-system access
  - ✓ maximize the sizes of written chunks
  - ✓ use derived datatypes for non-contiguous access
- If you are patient, read MPI standards, MPI-2.x or highier or libraries (based on MPI-I/O) like HDF5 or NetCDF

Useful links

- ✓ MPI – The Complete Reference vol.2, The MPI Extensions (W.Gropp, E.Lusk et al. - 1998 MIT Press)
- ✓ Using MPI-2: Advanced Features of the Message-Passing Interface (W.Gropp, E.Lusk, R.Thakur - 1999 MIT Press)
- ✓ Standard MPI-3.x: http://www.mpi-forum.org/docs
- ✓ The HDF Group Page: http://hdfgroup.org/
- ✓ HDF5 Home Page: http://hdfgroup.org/HDF5/
- ✓ HDF tutorial: http://hdf.ncsa.uiuc.edu/HDF5/doc/Tutor
- ✓ corsi@cineca.it: http://www.hpc.cineca.it
- ✓ ...practice practice practice

Acknowledgments

- ✓ Luca Ferraro
- ✓ Francesco Salvadore